SOURCE ENGINE FOR TRANSFERRING A RESOURCE TO LOCAL SECURE CACHE

Technical Field

The invention relates generally to the field of installing and updating computer program code in a computing system. More particularly, the invention relates to a system service running in a local system for saving program code installation files, using the files for updates and repairs, and controlling the downloading of setup, updates, and error reports.

Background of the Invention

The installation of program code in a computing system requires a the original program code to be loaded into the computing system from a CD or DVD, or downloaded over the internet to the computing system, or downloaded from a shared source in a corporate network to the local computing system. Subsequent updates may again require access to the original program to install the update. Unfortunately, the original program code is not always available as the user may have not kept the original CD or not kept the download authorization for the original program code.

It is with respect to these considerations and others that the present invention has been made.

Summary of the Invention

In accordance with the present invention, a method is provided for retrieving a program code resource from a source of program code and placing the resource in secure cache in the computer system. A user profile is impersonated by the computing system so that the computing system follows the user profile during retrieval of the resource. A source for the resource is picked, and the resource is opened. Whether the user has access to the resource is tested based on

the user profile. If the user has access, the resource is read to the secure cache. The resource is written into the secure cache and verified that it is the same as resource in the source. If the user does not have access, a new source for the resource is picked, and the resource in the new source is opened and read to the secure cache if the user has access to the resource in the new source.

In accordance with other aspects, the present invention relates to a source engine system in a computing system for retrieving a program code resource from one of a plurality of sources of program code. An open resource module in the source engine opens a program code resource in a source if the user has access. A pick source module picks a second source for the resource program code if the user does not have access. A read module transfers the program code resource to a writer module for a secure cache in the computing system. The writer module writes the program code resource into the secure cache. A verify module verifies the program code resource in secure cache matches the program code resource in the source, and a delete module deletes the program code resource from secure cache if the program code resource in secure cache does not match the program code resource in the source. The source engine system profile impersonates the user profile during retrieval of the program code resource and returns to the system profile after the program code resource is written to secure cache.

The invention may be implemented as a computer process, a computing system or as an article of manufacture such as a computer program product or computer readable media. The computer program product may be a computer storage media readable by a computer system and encoding a computer program of instructions for executing a computer process. The computer program product may also be a propagated signal on a carrier readable by a computing system and encoding a computer program of instructions for executing a computer process.

These and various other features as well as advantages, which characterize the present invention, will be apparent from a reading of the following detailed description and a review of the associated drawings.

Brief Description of the Drawings

FIG. 1 illustrates one embodiment of the invention where the source engine in a local computing system retrieves a resource and places it in secure cache for use by an installer module.

FIG. 2 illustrates an example of a suitable computing system environment on which embodiments of the invention may be implemented.

FIG. 3 illustrates the main operational flow of the source engine for retrieving resource information.

FIG. 4 shows the operational flow of operations performed by the cache resource module **306** in FIG. 3.

FIG. 5 illustrates the operational flow of operations performed by the reader module **414** in FIG. 4.

FIG. 6 shows the operational flow of operations performed by the writer module **412** in FIG. 4.

FIG. 7 shows the operational flow of operations performed by the extractor module **416** in FIG. 4.

Detailed Description of the Invention

Fig. 1 illustrates one embodiment of the invention having a source engine in a local computer. The local computer **100** includes the source engine software **102** as well as a secure cache system **104** and an installer **106**. Source engine **102** has exclusive control over acquiring software resources and loading them into secure cache **104**. The software resources would be retrieved by the source engine from a CD or a DVD drive **108** and written into the secure cache **104**. Alternatively, source engine **102** might retrieve the program code resource as a download over the Internet **110** from a web site supplying the program code resource. Yet a third possible source of program code would be a corporate network where the corporation has a license to

share the program code to local computers on the network. In this instance the source engine retrieves the resource program code from the corporate network **112** and again loads it into secure cache **104**.

When the program code is to be installed in the local computer's system then the installer module **106** asks the source engine **102** the location of the program code and secure cache **104**. The installer then reads the program code from the secure cache. The installer module may be a setup program. It may be a repair program for installing patches or an update routine to install updates, or it may be an alert system for installing repairs and updates.

It is significant that the installer module **106** does not have access to the secure cache except to read the program code placed there by the source engine. The source engine and the secure cache work together to provide a secure system for the original program code. This prevents inadvertent or deliberate alteration of the original program code retrieved by the source engine and loaded into the secure cache. The operations performed by the source engine to cache a resource in secure cache are shown in Figs. 3-7.

FIG. 2 illustrates an example of a suitable computing system environment on which embodiments of the invention may be implemented. In its most basic configuration, system includes at least one processing unit **202** and memory **204**. Depending on the exact configuration and type of computing device, memory **204** may be volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some combination of the two. This most basic configuration is illustrated in FIG. 2 by dashed line **206**.

In addition to the memory **204**, the system may include at least one other form of computer-readable media. Computer readable media can be any available media that can be accessed by the system **200**. By way of example, and not limitation, computer-readable media might comprise computer storage media and communication media.

Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer

readable instructions, data structures, program modules or other data. Memory **204**, removable storage **208** and non-removable storage **210** are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can accessed by system **200**. Any such computer storage media may be part of system **200**.

System **200** may also contain a communications connection(s) **212** that allow the system to communicate with other devices. The communications connection(s) **212** is an example of communication media. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. The term computer readable media as used herein includes both storage media and communication media.

In accordance with an embodiment, the system **200** includes peripheral devices, such as input device(s) **214** and/or output device(s) **216**. Exemplary input devices **214** include, without limitation, keyboards, computer mice, pens, or styluses, voice input devices, tactile input devices and the like. Exemplary output device(s) **216** include, without limitation, displays, speakers, and printers. Each of these "peripheral devices" are well know in the art and, therefore, not described in detail herein.

In the description of the source engine, which follows in reference to FIGS. 3-_, the logical operations of the various embodiments of the present invention are implemented (1) as a

sequence of computer implemented acts or program modules running on a computing system and/or (2) as interconnected machine logic circuits or circuit modules within the computing system. The implementation is a matter of choice dependent on the performance requirements of the computing system implementing the invention. Accordingly, the logical operations making up the embodiments of the present invention described herein are referred to variously as operations, structural devices, acts or modules. It will be recognized by one skilled in the art that these operations, structural devices, acts and modules may be implemented in software, in firmware, in special purpose digital logic, and any combination thereof without deviating from the spirit and scope of the present invention as recited within the claims attached hereto.

In Fig. 3 the operational flow for retrieving resource information is illustrated. The flow begins when request operation **302** receives a request from the calling program for resource information. Request operation requests the information from the source engine. The source engine in operation **304** detects whether the download exists. If the download containing the resource information exists, then the operation flow branches YES to the cache resource module **306**.

Cache resource operation **306** will see that the resource is cached, verify the resource and make the resource available. When the resource is available, the get operation **308** provides the resource to the calling program and the operation flow returns to the main flow of the source engine.

If the download does not exist, then it must be created in accordance with parameters from the calling program. In this situation the operation flow branches NO from the download test operation **304** to the create download package operation **310**. Create operation **310** receives the parameters of the download package from the calling program and builds the package. With the package defined and given an ID, the operation flow proceeds to create source operation **312**. Create source operation **312** receives parameter information defining the sources from the calling program and creates the sources IDs from which the resource information can be found. As

discussed above there will typically be multiple sources for the same resource information. After create source operation **312,** create resource operation **314** receives the list of resources defined by the calling program and generates resource IDs for the resource information so that the resources may be retrieved from a source. The completion of operations **310**, **312**, and **314** completely define the download, which is given an ID, and is then provided to the cache resource operation **306**.

The operational flow of the cache resource module is illustrated in Fig. 4. The operation flow begins at mark operation **402**, which marks the retrieval of the resource information as pending. Cache test operation **404** then tests whether the resource is in cache or must be placed in cache. If the resource is in cache, then operation flow branches YES to verify resource operation **406**. Resource information is verified based on a check phrase written with the resource information in the cache. This check phrase is a hash number computed from the original resource information using a hash algorithm. A hash number for the resource information in cache is calculated according to the same hash algorithm. If the calculation produces a hash number matching the check phrase, then the resource in cache is verified.

Resource verify test **408** detects whether the verification operation **406** was successful. If the resource was verified, then the operation flow branches YES to mark operation **412** to mark the resource available from cache. The program flow then returns to get resource information operation **308** in Fig. 3. If the resource information cache did not verify, then the operation flow branches NO to delete operation **410**. Delete operation **410** erases the written resource information in cache, and the operation flow proceeds to operations to rewrite or to load the resource information in cache. Operations loading the resource information in cache are described hereinafter.

Returning to the in cache test operation **404**, if the resource information is detected as not in cache, then the operation flow branches NO to the load cache operations. The load cache operations amount to two parallel operations: (1) reading resource information from a source and

(2) writing the resource information into cache. The writing operation is performed by writer module **412**, which is described hereinafter with reference to Fig. 6. The retrieving of the resource information from a source can be accomplished in either of two ways depending on whether the resource is stored at a first level in a source or whether it is stored at a second level within a source. By second level it is meant that the resource is inside a container stored in the source. The container is often referred to as a cabinet file and the resource as a file inside the cabinet file. If the resource is in a resource file in a source, then it is retrieved by reader module **414**, which is described hereinafter with reference to Fig. 5. If the resource file is in a cabinet file or a folder in the source, then the resource is retrieved by extractor module **416**. Extractor module **416** is described hereinafter with reference to Fig. 7. The detection of whether the resource file is in a source or in a cabinet in a source is detected by contained test **418**. Contain test **418** branches the operation flow to the extractor module **416** if the resource file is contained within a file in the source. If the resource file is not contained in another file, then the operation flow branches NO from contain test **418** to the reader module **414**.

In Fig. 5 the operational flow of the reader module begins with the initialize operation **502**. Initialize operation **502** loads all the necessary parameters for the reader to retrieve the resource from a source. After initialization, impersonate operation **504** loads all of the personal parameters for the user into the retrieval process. In effect the impersonate operation replaces the system profile with the user's profile. Thus, the retrieval process is operating based on the profile of the user rather than the profile of the computing system that the user is using. This is important as it provides additional security, i.e., only a user with the user's profile may perform the cache loading and the retrieval of the resource from the cache.

After the reader is initialized, and the user's profile is loaded, then source identified test **506** detects whether the user has specified a source from which to retrieve the resource. If the user specified a source for the resource, then the operation flow branches YES from source identified test to open resource operation **508**. Open resource operation gets the resource file

from the source and opens it if it is found and if the user has access. Resource open test **510** is checking to see that the resource was found and could be opened. Access test **512** is checking whether the user has access to the resource, i.e., is identified as a user that can have the resource. If resource open and access tests are passed successfully, then the operation flow branches YES from access test **512** to read data operation **514**. Read data operation **514** begins the process of transferring the data to the writer. As each block of data is read, put data operation **516** transfers the data into the writer, and more data test operation **518** tests whether more data is in the resource file that is being written to cache. If there is more data, the operation loops back to read data operation **514** and this loop **514, 516, 518** continues until all the data of the resource file is read to the writer. As will be discussed hereinafter, while this data is being read to the writer the writer is writing it and verifying it in the cache. After all of the data from the resource file is read, then the operation flow returns to the flow in Fig. 3.

The above operation flow is the simplest and most direct flow through the reader module illustrated in Fig. 5. However, if the resource was not found or would not open or the user did not have access to the resource in the source specified by the user, then the operation flow branches NO from access test **512** or NO from resource open test **510** and proceeds to more sources test **520**. If there are more sources identified by the download package for the resource, then pick operation **522** picks another source for the resource. The sources will be prioritized in the download package and the pick operation **522** will accordingly pick the next possible source on the priority list. The operation flow then proceeds to operation **508** and attempts to open the resource in this new source.

If all the sources have been tried and either the resource was not found or could not be opened or the user had no access, then more sources test **520** will fail. The operation flow will branch NO from the more sources test **520** to the send operation **524**, which will send an error message. The error message will indicate the cause of the problem. The resource was not found, the resource would not open, or the user does not have access.

While the reading of data from the resource is occurring, the writer module is operating in parallel to load the data into cache. Fig. 6 shows the operational flow of the writer module. The write operation begins with initialize operation **602**. Initialize operation **602** initializes the writer according to the parameters received with the download package. After initialization the writer at operation **604** waits for data from the reader or extractor. When data is received more data test operation **606** detects whether there is more data in the resource information than what has been received at present. If there is more data, then operation flow branches YES to the write data operation **608** that begins to write the data into cache. Thus, while the reader or extractor is reading information, the writer module is writing the data into cache as soon as it is received. This loop of operations **604**, **606**, and **608** continue until the more data test operation **606** detects that all the data in the resource has been received. When this occurs, the operation flow branches NO to the mark operation **610**. Mark operation **610** indicates to the user that the caching phase of the operation has been successful. After the resource information is in cache then verify operation **612** verifies the resource information. Verify operation **612** operates in the same manner as described above for the verify operation in Fig. 4. In other words, if the hash count for the resource information matches the hash count downloaded when writing resource into cache and the resource is verified. Verified test operation **614** tests whether the resource information was successfully verified. If it was not, then the operation flow branches NO to delete operation **616**. Delete operation **616** deletes the resource information just written into cache and the operation flow returns to the source engine.

If the verify resource operation is successful, the operation flow branches YES from verified test **614** to mark operation **618**. Mark operation **618** marks the resource information as available from cache. Un-impersonate operation **620** then removes the user profile from the retrieval process so that the process will return to operating under its computing system profile rather than the user's profile. Operation flow then returns to the get resource information operation in Fig. 3.

If the resource information had been retrieved using the extractor module, the same writing module operation flow occurs. However, the reading of the resource information proceeds as indicated in Fig. 7. Fig. 7 shows the operational flow for the extractor module. In the extractor module the operation flow begins at initialize operation **702** that initializes the extractor according to the parameters received with the download package. Open operation **704** then opens the container in the source that contains the resource file. If the opening of the container is not successful as detected by test operation **706**, the operation flow will branch NO and send and error message to the user. An alternative source will then be tried if possible. If the container is opened successfully, then the operation flow branches YES from test operation **706** to impersonate operation **710**.

Impersonate operation **710** impersonates the user in the same manner as impersonate operate **504** does in the reader module as shown on Fig. 5. Basically, the impersonate operation applies the profile of the user to the source engine so that the opening of the resource and the security check will be performed in accordance with the user's profile.

Open resource and security check module **712** operates in the same manner as discussed above for opening the resource and checking user access as described in Fig. 5. If the resource is found, if it opens successfully, and if the user has access, then operation flow will proceed to the read out module **714**. Read out module **714** operates in the same manner as operations **514**, **516**, and **518**. The extractor module operates in parallel with the writer module so that data is retrieved from a source and read into cache in parallel operations. The extractor module is similar to the reader module with the additional operations necessary to open a container that contains the resource file.

The various embodiments described above are provided by way of illustration only and should not be construed to limit the invention. Those skilled in the art will readily recognize various modifications and changes that may be made to the present invention without following the example embodiments and applications illustrated and described herein, and without

departing from the true spirit and scope of the present invention, which is set forth in the following claims.